INTELLIGENT JAMMING USING DEEP Q-LEARNING

Noah Thurston, Garrett Vanhoy, and Tamal Bose
University of Arizona Department of Electrical and Computer Engineering
Tucson, AZ, 85719
noahthurston, gvanhoy, tbose@email.arizona.edu

Faculty Advisor:
Garrett Vanhoy

ABSTRACT

The threat of a malicious user interfering with network traffic so as to deny access to resources is an inherent vulnerability of wireless networks. To combat this threat, physical layer waveforms that are resilient to interference are used to relay critical traffic. These waveforms are designed to make it difficult for a malicious user to both deny access to network resources and avoid detection. If a malicious user has perfect knowledge of the waveform being used, it can avoid detection and deny network throughput, but this knowledge is naturally limited in practice. In this work, the threat of a malicious user that can implicitly learn the nature of the waveform being used simply by observing reactions to its behavior is analyzed and potential mitigation techniques are discussed. The results show that using recurrent neural networks to implement deep Q-learning, a malicious user can converge on an optimal interference policy that simultaneously minimizes the potential for it to be detected and maximizes its impediment on network traffic.

INTRODUCTION

With the increasing availability of relatively cheap software-defined radios (SDRs), the number and complexity of security threats at the physical layer of wireless networks has also increased. On the passive side of security threats is the development of algorithms that can exploit information that cannot be concealed simply by using encryption such as symbol-rates and packet sizes to infer the nature of a user's activity. These passive security threats, in which the malicious user does not actively transmit, ultimately enable higher levels of intelligence for the active security threats such as jamming or spoofing. For example, an intelligent jammer can classify the modulation of a frame and choose to selectively interfere with some frames and not others to simultaneously impede network throughput and avoid detection.

However, many of the intelligent jamming techniques may be difficult to implement in practice due to the need for the malicious user to detect and react to traffic on the order of microseconds. Hence, to combat this threat, physical layer waveforms can be designed to minimize detectability by an eavesdropper such that by the time the eavesdropper detects a frame, it no longer has time to react. A more effective threat than this reactive approach, is an intelligent jammer that can predict the behavior of network traffic through both passive observation and through observation of reactions

to actions taken by the intelligent jammer. Such a jammer would not be limited by the need to react in such short times or perhaps even need to detect frames and thus could likely be implemented on far less capable hardware.
The aim of this work is to examine some aspects of such an approach using the same modern reinforcement learning techniques that have received attention for being able to reliably defeat even the world's best in some video or board games. By adopting a vastly simplified model for some common communication protocols such as 802.11, several reinforcement learning techniques are tasked with learning the optimal strategy for impeding network traffic, while minimizing its exposure through transmissions. Comparing the performance of these reinforcement learning techniques with more simple jamming techniques as a baseline, the results show that it is possible for a reinforcement learning algorithm to learn the optimal policy to jamming simply by observing reactions to its own actions.

BACKGROUND

Basic Jamming Agents

Jammers can be sorted into one of five different categories; constant, intermittent, reactive, adaptive, and intelligent [1]. These jammers vary in respect to the simplicity of their implementations, their effectiveness, their power usage, and their ability to remain undetected.
Constant jammers are very simple in that they output a constant signal of modulated or unmodulated noise with the intent of degrading signals or occupying bandwidth. These jammers are very effective, but at the cost of high energy usage. They can also be easily detected by statistical analysis of received signal strength (RSS), carrier sensing time (CST) and packet error rate (PER).
Intermittent jammers are similarly simple to their constant counterparts, but behave by producing signals at random or predetermined intervals. This can greatly reduce their energy usage, but also lower their effectiveness as jammers. These too can be detected with analysis of the RSS, CST and PER.
Finally, intelligent jammers act by exploiting weaknesses in the upper-layer protocol. Often this is accomplished by targeting and degrading packets that are necessary to synchronize radio users, instead of payload packets. This has been shown to be effective when exploiting the IEEE 802.11 standard, commonly known as Wi-Fi, by disrupting the transmission of request to send (RTS) and clear to send (CTS) packets. These packets are often much shorter than the proceeding payloads, but by jamming these control packets the jammer can ensure that the payloads will not even be sent. The strength of this intelligent approach is that much less energy is used because small control packets are targeted instead of much longer data payload packets. They are also unlikely to be detected by analysis of RSS, but could be found when observing changes in CST and PER. Furthermore, as suggested by the name, these jammers are much more complex than the constant and intermittent designs and require knowledge of the protocol being used beyond the physical layer. This work focuses on the implementation and evaluation of constant, random, and intelligent jammers.

Existing Intelligent Jamming Techniques

The current research leveraging machine learning to jam wireless systems revolves around the network layer. This involves observing, classifying and attacking packets when the attacker believes that the corruption of those packets in particular will be more detrimental to the transmission of data than others. Often it is because these packets are heavy in control information that sync transceivers in the network and work to setup communication before the payload is sent.

This network layer focus is demonstrated in [2] by Amuru and Bueuhrer, where they model and attack the RTS-CTS handshake system of an 802.11-type wireless network. This is done by framing the problem as a Markov decision process, and using the current state of the network to determine the best action to be taken. This simulation is useful to see the effects the time delay has on the attacker's ability to learn, but it relies on the assumption that the protocol used by the transmitter-receiver pair is known. It also depends on the use of ACK/NACK packets, which may not be used by a given protocol, or may not be available to jammers in certain situations.

In [3] a similar approach is taken, in which packets are jammed and ACK/NACK packets are monitored for feedback. However, the entire target packet is not jammed. Instead, a pulse jamming ratio ρ is used to stochastically raise and lower the strength of the jamming signal. The benefit of this approach is that the jammer can operate more efficiently and decrease its likelihood of being detected.

The "Greedy Bandit" algorithm was proposed by ZhuanSun et. al. to improve upon the "Jamming Bandits" algorithm [4]. They address the need to learn without ACK/NACK packets and propose two possible behaviors that can be observed in place of them; changes in power and enduring time. This takes advantage of the concept that if the jammer is successful in interfering with the communication of the sender-receiver pair, that the power of the signals transmitted will increase, and the pair will change their communication parameters quickly. We will continue this line of thinking and use modulation encoding changes as a way to detect whether or not jamming was successful.

These past papers have succeeded in using machine learning to jam packet level network communications. They have demonstrated how Markov decision processes are useful for determining optimal actions using state information. However, the behavior of wireless networks is not only dependent on the current state of the network, but also the previous states. One example of this behavior is when transmitter-receiver pairs are determining the modulation encoding scheme to be used during transmission; a modulation will be chosen such that bits per symbol will be maximized without transmission failing regularly due to channel noise. And if the channel deteriorates during transmission such that many packets are failing, a lower modulation can be chosen to decrease the number of failing packets.

This behavior is just one example of how the network's actions are dependent on data from many time steps in the past, and not solely based on the current state of the network. Furthermore, this paper seeks to demonstrate how an agent wishing to optimally disrupt communications between a sender-receiver pair must be able to leverage past state information in their decision making. We show that this can be accomplished using deep Q-learning paired with recurrent neural networks.

METHODOLOGY

Epsilon Greedy

Epsilon greedy is a common and simple implementation of a reinforcement learning algorithm to solve a multi-armed bandit problem. Multi-armed bandit problems occur when an agent must decide how to allocate a finite number of decisions or resources among several competing choices. At every time step, a choice must be made as to which action should be taken. With a probability of ϵ, where $\epsilon \in [0, 1]$, a random action is taken. With a probability of $1 - \epsilon$ the action that has returned the highest average reward is chosen. This algorithm works well when one action consistently returns higher rewards than the others, and the environment in which the agent is in is static.

Markov Decision Processes

Markov decision processes were first described by Richard Bellman in 1950s [5]. They consist of a finite number of states, and a set of actions that can be taken by an agent at each state. When each action is taken, there is a probability that the environment will move to a certain new state, or stay in the current state. Each transition, either to a new state or continuing to stay in the current state results in some reward.
The goal of the agent is to maximize the rewards it receives at the current step and in future steps. To do this, the value of each state must be calculated, and this is done with the Bellman Optimality Equation. Shown below, it describes the value of a state as V^* which is equal to the weighted average reward earned by taking the best action plus the discounted value of the state transitioned to:

$$V^*(s) = max_a \Sigma_{s'} T(s, a, s')[R(s, a, s') + \gamma \cdot V^*(s')] \qquad \text{(Bellman Optimality Equation)}$$

- $T(s, a, s')$ is the probability that the transition from state *s* to *s'* occurs given action *a* is taken.
- $R(s, a, s')$ is the reward the agent receives as it transitions from *s* to *s'* given action *a* is taken.
- $\gamma \in [0, 1]$ is the discount rate that decreases the value of future rewards.

However, in order for the agent to take optimal actions, it must know the value of taking an action given its current state. This value is a state-action pair, also known as a Q-value. The Q-learning algorithm allows for an agent to discover the Q-values of all state-action pairs through experimentation, without knowing the transition probabilities or the rewards beforehand.

$$Q_{k+1}(s, a) \leftarrow (1 - \alpha)Q_k(s, a) + \alpha(r + \gamma \cdot max_{a'} Q(s', a')) \qquad \text{(Q-Learning Algorithm)}$$

- $\alpha \in (0, 1)$ is the learning rate.
- r is the reward earned when leaving state *s* after taking action *a*.

Approximate Q-Learning

The Q-learning algorithm is successful in small state spaces, but can become intractable as the number of possible states and actions grow. The solution to this problem is Approximate Q-Learning. This is when a function with a manageable number of parameters is used to estimate Q-values either with information about the state or with expert features. It has been shown that neural networks are apt at representing these functions to estimate Q-values, a technique called Deep Q-learning [6].

Deep Q-learning is accomplished by creating a neural network whose input is the current state, and whose output is the Q-values of all possible actions that can be taken at the current state. Training is conducted by having a neural network labeled "actor" interact with the environment by calculating the Q-values of each action at a state and then either picking the optimal action with a probability of $1 - \epsilon$ or exploring non-optimal actions with a probability of ϵ, where $\epsilon \in [0, 1]$.

These Q-value estimations, the actions taken and the resulting reward are recorded into a replay memory. Then after a set of actions has been stored in the replay memory, a "critic" neural network is used to estimate the Q-values of each state visited. Using this data, the deep Q-learning cost function is used to determine the degree to which the critic is incorrect in estimating the Q-values of each state.

$$J(\theta_{critic}) = \frac{1}{n}\Sigma_a(y - Q(s, a, \theta_{critic}))^2 \qquad \text{(Deep Q-learning Cost Function)}$$

- θ_{critic} and θ_{actor} are the parameters of the critic and actor neural networks, respectively.
- $y = r + max_{a'}Q(s', a', \theta_{actor})$ and is the targeted Q-value prediction as made by the actor.
- $Q(s, a, \theta_{critic})$ is the critic's prediction of the Q-values.
- *n* is the number of possible actions.

Notice that the cost function is the mean squared error of the target Q-values calculated by the actor and the predicted Q-values calculated by the critic. This error is used to optimize the critic neural network and on regular intervals the parameters of the improved critic network are copied to the actor.

Classic and Recurrent Neural Networks

Using Deep Q-Networks is a powerful approach to Markov decision processes, and its relies on neural networks. The most basic neural network is comprised of layers of nodes. Each node multiplies an input x_i by a weight w_i and then adds a bias b. Every node has a weight corresponding to each input from the previous layers nodes, but only a single bias. The sum of the node's bias and the products of all respective weights and inputs is then passed through an activation function, $\sigma(n)$ and outputs y.

$$y = \sigma(w \cdot x + b) \qquad \text{(Single Node Output Function)}$$

The output of each node then becomes an input for every node in the next layer, or becomes the final output of the network. After a network has output a value, it can then be compared to the

theoretical output and a cost function like MSE is used to calculate the error. After the error is found, the gradient of the error with respect to the weights and biases is then calculated. This allows for the weights to be moved along that gradient to reduce the error in the output. When the network is trained on individual steps in this way it is known as stochastic gradient descent [7].
Recurrent neural networks are similar to their classic counterparts but are modified to handle time series data. At each time step, nodes calculate an output y_t as well as a hidden state h_t.

$$h_t = \sigma(w_{hx} \cdot x_t + w_{hh} \cdot h_{t-1} + b_h) \qquad \text{(RNN Hidden State Function)}$$

$$y_t = \sigma(w_{yh} \cdot h_t + b_h) \qquad \text{(RNN Output Function)}$$

- w_{hx} is the matrix of weights connecting the input and hidden layer.
- w_{hh} is the matrix of weights connecting hidden layers of adjacent time steps.
- w_{yh} is the matrix of weights connecting the hidden layer and the output layer.

Because the output y_t is a function of both the inputs x_t and the previous hidden states h_{t-1}, the network can make calculations based on data from both current and previous time steps. This gives the network the ability to handle time series data. Training RNNs works similar to training classic neural networks, but the back propagation algorithm is modified to account for the numerous time steps.
The specific architecture of RNN used in this paper is the Long Short Term Memory design (LSTM). It functions similar to the previously described RNN, but the hidden layer is replaced by a "memory cell" that is better equipped to handle long time series data. Further review of RNN architectures can be found in [8].

PROBLEM FORMULATION

Environment Design

OpenAI Gym is a library that allows for the creation of environments such as games as well as for agents to act in these environments. The power of this library comes from developers' ability to create and share environments that allow other developers to create agents for them. As such, this framework was chosen to create a game called "Throttle" in which the agent acts as an eavesdropper (Eve) between a sender-receiver pair.
The sender-receiver pair chooses an arbitrary modulation encoding of 0, 1, 2, or 3, with a higher number representing a higher encoding and higher data throughput. Each time step, a single packet is sent across the channel and Eve observes the modulation, which it interprets as the state of the environment, and can decide with what power to jam the signal; 0, 1, 2, or 3, with 3 being the highest power. A jamming power of 2 and 3 results in the packet to fail. If four consecutive packets fail, the sender reduces the modulation by one. If four consecutive packets succeed, the sender increase the modulation by one. After each action is taken by Eve, it is rewarded using the Reward Equation:

$$r(a) = \begin{cases} 4 - a & : s \leq 1 \\ 0 & : s > 1 \end{cases} \qquad \text{(Reward Equation)}$$

As shown, Eve is rewarded solely for keeping the modulation (known as the state s) at 0 or 1, and not by the effectiveness of its jamming. Eve does, however, receive a lower reward for jamming (taking action a) with high power, and is therefore motivated to keep the modulation below the threshold with as little jamming and as low jamming power as possible.

Agent Designs

Six different agents were designed and tested in the "Throttle" environment. At every time step, the agents were told which of the 4 modulations were being used and were given the ability to decide the jamming power.

The first two agents were constant and random jammers. By their design, neither made use of the state information they observed, and instead had hard-coded behaviors. The constant jammer always used the max jammming power of three at every time step. The random jammer chose a jamming power between zero and three by sampling from a discrete uniform probability distribution. This was implemented using numpy's randint function.

For the epsilon greedy agent, each of the four jamming powers were considered an arm that could be taken. The epsilon value was set at 0.25.

The fourth agent was implemented with approximate Q-learning by having a neural network predict the Q-values, thus creating a deep Q-network (DQN). The current modulation was input into the network and the output was the four Q-values, each representing the predicted Q-value of the four different jamming powers that could be chosen. The neural network had a hidden layer of 16 nodes each with a ReLU activation function, and then an output layer of four nodes.

The fifth agent was similar to the previous DQN except a recurrent neural network (RNN) was used to predict the q-values. The input was the last six time steps of the state, inclusive of the current time step, which is the modulation encoding used by the sender-receiver channel. The hidden layer had 16 nodes which was unrolled in time over the six time steps. The output of the node in the last time step of the hidden layer went to a layer of four non-RNN nodes which output the predicted q-values.

The final agent designed also used a RNN but each input was the current state and the last action taken. This made the input size two for each of the six time steps. The hidden layer had the same architecture of 16 hidden nodes then an output of four non-RNN nodes.

Model Training

The epsilon greedy agent as well as the DQN agents all required training in order to maximize their average reward. All models were trained for 5,000 time steps, but the epsilon greedy model had its epsilon fixed at 0.25 while the DQNs used a dynamic epsilon value. Their value began at 1.0 and decreased rationally down to 0.01 over 5,000 time steps. This was necessary in order for the models to be able to thoroughly explore in the beginning, but then almost always choose the optimal action towards the end of training. Without the decreasing epsilon value, the DQNs with RNNs would not observe successive time steps of optimal behavior and could not learn the optimal behavior of jamming only once every four time steps.

As described previously, mean squared error was used to define the cost function. To optimize the

network, Adaptive Moment Estimation (Adam) was used. Adam is a form of Stochastic Gradient Descent (SGD) that uses a combination of past first and second moments to calculate the training step [7]. This gives the training algorithm "momentum" that moves it past local extrema which can often keep RNNs from converging on global extrema.

RESULTS

Figure 1: Summary of Agent Behaviors

Agent	Observations	Actions Taken	State Behavior (Modulation)	Average Reward
Constant Jammer	None	Jams constantly at 3	Constant at 0	1
Random Jammer	None	Jams randomly	Alternates between 0 and 1	1.22
Epsilon Greedy	None	Jams constantly at 2	Constant at 0	2
DQN with Neural Network	Current modulation	Jams with power 0 in modulation 0, jams with power 2 in modulation 1	Alternates between 0 and 1	3
DQN with Recurrent Neural Network	Current and past modulations	Jams with power 0 in modulation 0, jams constantly after 6 time steps in modulation 1	Alternates between 0 and 1	3.25
DQN with Recurrent Neural Network	Current and past modulations and actions	Jams once with power 2 every 4 time steps (optimal behavior)	Constant at 0 or 1	3.5

The summary of the results of each type of agent can be seen in Figure 1. As a baseline, constant and random jammers were first tested in the throttle environment. The constant jammer performed the worst although it completely prevented network traffic because it also maximally exposed itself through jamming. The random jammer did slightly better than the constant jammer because it only occasionally exposed itself through jamming, but also often successfully jammed frames.

Epsilon Greedy Agent This agent was trained for 10,000 time steps, with an epsilon value of 0.25. The agent was able to settle on the jamming power of two being the optimal power, and once it did the average reward of the arm approached two. A power of two is optimal for the agent because it fails packets in order to keep the modulation and coding at one or lower, but it also maximizes reward. These results demonstrate that the agent was able to learn that the power of two was powerful enough to fail packets and that three was unnecessary and only lowered the reward earned.

DQN Agent This agent demonstrated significant improvements over the epsilon greedy agent. It succeeded in learning to use different jamming powers when observing different modulation encodings. When it observed an encoding of zero, the agent jammed with a power of zero and reaped a max reward of four. After four time steps, the environment would then increase the encoding to one, and observing this new state the agent would jam with a power of two. It did this consistently while in state one, earning a reward of two until the modulation was degraded back down to zero. This resulted in an average reward of three. Although this reward is higher than those earned by previously tested agents, it still shows how the agents lack of memory of the past states and actions led to it jamming several packets in succession when it didn't need to.

DQN with RNN Agents Next a DQN was constructed similarly to the previous one, except with

Figure 2: Behavior of NN-DQN Observing the Current State

Time Step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Modulation	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
Jamming Power	0	0	0	0	2	2	2	2	0	0	0	0	2	2	2	2
Reward	4	4	4	4	2	2	2	2	4	4	4	4	2	2	2	2

its network composed of recurrent nodes. The agent reacted consistently and followed a pattern of 16 time steps shown in figure 10. In time steps one through four, the state remained at zero and the agent jammed with a power of zero earning the max reward of four. In time step five the encoding modulation was increased to one and the agent jammed with a power of two in order to fail the packet. Then for the next three time steps the agent jammed with a power of zero because it was able to make its decision with six time steps of data, which included a transition that it knows it always jams. Then, in time step nine it knew it had to jam again to keep the modulation from increasing to two. However, it continued to jam because all previous states it saw (the current state and the previous five) were all of a modulation encoding of one and it could not remember which action it took during those states. This continued until the modulation was decreased back to zero in time step 13. In the final time step of the series, number 16, the agent jammed with two knowing it had not jammed the previous three time steps since the modulation transition from one to zero. After time step 16, the pattern repeats from time step one again.

This behavior demonstrates the network's ability to leverage the past state observations, but also how when the state is constant for more time steps than the agent remembers, it acts sub-optimally. This is because it is unable to deduce what its past actions were, and thus whether it is necessary to jam or not.

The final DQN was constructed with an RNN but was given both the currently observed modulation encoding and the previously taken action, for six time steps at a time. This resulted in the modulation encoding remaining constant at one, and the agent jamming with a power of two once every four time steps. This behavior is optimal for the given environment, and earns the agent an average reward of 3.5. It also demonstrates how in order for the RNN to act optimally it must be able to make decisions based on both the environments past states and the agents own past actions.

Figure 3: Behavior of RNN-DQN Observing States

Time Step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Modulation	0	0	0	0	1	1	1	1	1	1	1	1	0	0	0	0
Jamming Power	0	0	0	0	2	0	0	0	2	2	2	2	0	0	0	2
Reward	4	4	4	4	2	4	4	4	2	2	2	2	4	4	4	4

Figure 4: Behavior of RNN-DQN Observing States and Actions

Time Step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Modulation	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Jamming Power	0	0	0	2	0	0	0	2	0	0	0	2	0	0	0	2
Reward	4	4	4	2	4	4	4	2	4	4	4	2	4	4	4	2

CONCLUSIONS

Jamming continues to be a threat to the security of wireless communications and mitigation techniques must evolve together with the complexity of threats. Currently described intelligent jammers work to exploit upper layer protocols to decrease sender-receiver throughput, but are difficult to implement in practice. We've demonstrated using modern machine learning techniques that common adaptive modulation and coding schemes can be exploited in a predictive manner rather than a reactive manner.

The results of this work admittedly only scratch the surface of what an intelligent jammer equipped with modern machine learning techniques can potentially accomplish. Since many wireless protocols obey a common set of rules, they can potentially be learned through reinforcement learning techniques over a short enough time. Dynamic modulation and coding is only one such example, but it's logical to see how other practices such as handshakes, ACK/NACK packets, and static packet structures are predictable and possibly open for exploitation. The defense against this attack being that radios need to constantly change their behavior in order to deny an intelligent attacker the time to learn the pattern they are following.

References

[1] Y. Zou, J. Zhu, X. Wang, and L. Hanzo, "A Survey on Wireless Security: Technical Challenges, Recent Advances and Future Trends," vol. 104, no. 9, pp. 1727–1765, 2015.

[2] S. Amuru and R. M. Buehrer, "Optimal Jamming using Delayed Learning," *Proceedings - IEEE Military Communications Conference MILCOM*, pp. 1528–1533, 2014.

[3] S. Amuru, C. Tekin, M. Van Der Schaar, and R. M. Buehrer, "Jamming Bandits - A Novel Learning Method for Optimal Jamming," *IEEE Transactions on Wireless Communications*, vol. 15, no. 4, pp. 2792–2808, 2016.

[4] S. Zhuansun, J.-a. Yang, H. Liu, and K. Huang, "A Novel Jamming Strategy-Greedy Bandit," no. 2, pp. 0–4, 2017.

[5] A. Géron, *Hands-on Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2017.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[7] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016.

[8] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A Critical Review of Recurrent Neural Networks for Sequence Learning," pp. 1–38, 2015.